A DATA-FLOW LANGUAGE FOR SPECIFYING BUSINESS DATA PROCESSING APPLICATIONS

V. M. Malhotra and V. Rajaraman

Computer Science Program,
Indian Institute of Technology,
KANPUR (208 016) INDIA

A high level specification language called Business Data-Flow Language (BDFL) based on the data-flow model of computation is described in this paper. A translator to convert BDFL specifications into COBOL programs has been implemented. A novel feature of the proposed translation system is the use of Data Base Management System (DBMS) in conjunction with COBOL. This relieves the user of the need to define data structures for the intended data processing application.

Key words and phrases:

Business data processing, specification languages, data-flow languages, non-procedural languages, translation, DBMS.

CR categories:

3.50, 4.12, 4.22, 4.33, 4.34.

1. INTRODUCTION

The aim of this paper is to explore the use of functional style of programming [AGP78, AGP80, BAC78, DEN73] in specifying business data processing problems. As the specification language we intend to introduce is meant to be used by the business data processing community and the specification should be self-documenting the syntax adopted is similar to the more verbose form of functional languages, viz., data-flow languages, as opposed to the concise and mathematical syntax of FP and FFP [BAC78].

Apart from the syntactical similarities, the proposed specification language, henceforth called, Business Data-Flow Language (BDFL), has yet another feature resembling other data-flow languages. Both aim at describing asynchronous computational activities. Data-flow architecture assigns specified functions to different processors. Each processor works asynchronously and, co-operates to achieve the desired computation. In a similar way BDFL specifies the data processing activities of a set of 'independent' groups. Each group works asynchronously and cooperates to perform the required job. The difference between the two stems mainly from the intended user community and the aims of the language.

In section 2 of this paper a brief survey of the intended application area and the reasons that make the use of data-flow languages an attractive proposition for business data processing are presented. Section 3 is devoted to a description of the features and functions of the language. Section 4 presents a brief sketch of the translator that is used to translate the high level specification of the application into a program in a conventional programming language. This 'conventional language' is COBOL and makes use of a CODASYL DBMS implementation on DEC System 10 for the purpose of mass storage of data. COBOL was selected as it is currently the most popular language for business data processing. In section 5 an example and some strategies for enhancing the efficiency and the readability of the generated COBOL program are presented. In section 6 we conclude by presenting observations based on experience with our system.

2. DATA-FLOW LANGUAGE AS SPECIFICATION TOOL

To evolve and understand the language for specifying business data processing, it is imperative to examine the structure of business organisations and the manual methods they use for data processing. It has been observed [HHKW77] that, save a few minor exceptions, these organisations have the following characteristics:

"- Businesses are divided into organisational units such as divisions and departments. The structure of the data processing follows closely this organisation.

- These organisational units communicate with each other via documents in a stylised manner.

- Document manipulation rarely involves complex mathematical computations.

- Documents are frequently stored for later use and reference.

- Operations are frequently performed on groups of documents."

An important characteristic that has been missed in the above list is the asynchronous nature of processing in manual systems. The term asynchronous has been used to denote the absence of an explicit order or sequence in which functions and subfunctions are performed. A group can perform a data processing function any time after the arrival of all input 'data bags' it needs. Different groups can thus independently carry out their functions. Often a group may have to defer an action till certain time or date. Such a situation may be modelled as a group waiting for an (empty) data bag that arrives only at the stipulated time which would trigger the data processing activities.

It is thus evident that the hierarchical structure and data processing functions of an organisation can be specified by means of nested blocks of functions in a data-flow program. Data-lines mirror the flow of information and documents between the organisational units.

The use of data-flow mechanisms for specifying business data processing problems has already been made in the design of BDL [HHKW77]. Their approach differs from our work in numerous ways, the foremost being the need for a graphic terminal to write a BDL specification. The language BDFL does not require one. We also believe that even though prior declarations of document and file structures help in efficient execution of the specifications, they present difficulties in writing specifications. This is a serious impediment for non-programmers. We make extensive use of data base management system (DBMS) to remove the need for prior declarations of the record and file structures.

Furthermore, the explicit lines or paths that need be drawn to indicate flow of information in the BDL description may prove inconvenient. The severity of the inconvenience will be more evident when large amount of relatively trivial data is passed around in the organisation.

Specification languages, like MODEL [PPS79], we feel, are not suited for laymen. The statements in the language depict the relationships among the various quantities of interest to the application. A layman would often prefer to list the steps involved in arriving at the result, instead of listing all the relevant relationships that ought to be satisfied. In this respect, data-flow languages score over languages like MODEL by providing a syntax that permits the user some 'feel' for the procedural nature of the computations. What is more interesting is the fact that unless the perceived procedural description contains a gross error, it (the procedural trace) belongs to the set of valid traces. A trace is considered valid if a single processor executing the functions in the order they occur in the trace do not abort due to the absence of a required data.

A serious limitation of data-flow languages is their inefficient implementation on von Neumann machines. Though we do not present a solution to this general problem, yet we feel that it does not pose a serious drawback for the intended applications. The proposed language would reduce program development time and thus save valuable professional man-hours. As the trend is towards reduced hardware costs and increased scarcity of applications personnel the trade-off proposed is beneficial. Further, program transformations and other optimisation techniques may be used to reduce computer resource requirements. The use of pre-written code in the object language (COBOL) to replace frequently occuring combinations of functions in BDFL specifications, during translation process, may be made to improve the performance of the generated code.

3. THE BUSINESS DATA-FLOW LANGUAGE (BDFL)

The BDFL is a block structured, expression oriented functional language based on Id [AFG80, AFG78]. Data-lines are given names and they carry values from one function to others. Each data-line in the BDFL specification may be loaded with a value by one and only one function. An expression defining the function is written to the left of symbol --> and the list of data-line names receiving the values to its right. This construct is called a rule.

The data-lines in BDFL are not explicitly typed. It is the usage and the type of information that flows along a data-line that defines the type for the data-line in the specification. The data-types that have been identified for the purpose of current implementation are: null, integer, record, and dataset. The other data-types of interest to the application area are: real, character and string. These can easily be added to the current list of permitted data-types. In what follows we shall use the term scalar to denote the types integer, real, character and string. The coercion of a value of one type to a value of different type is assumed available and is performed automatically when permitted.

The value of type null (represented as () in BDFL specifications) can travel along any data-line irrespective of the inferred type for the data-line. With the exception of the dataset constructor and the test on the nullity, all atomic expressions and functions receiving a value of type null produce the result of type null. While the dataset constructor ignores the null value, the function NULL produces boolean value TRUE. For values of all other types the function NULL produces the boolean value FALSE.

The inclusion of record type in BDFL has been inspired by the tabular structure of ledgers and the stored data in manual data processing applications. A record is a collection of 'selector:value' ordered pairs. A selector is of type string whereas value is of type scalar.

The projection function, OF, is used to isolate the value associated with a given selector. Thus, a OF C returns the value B, if 'a:B' is an ordered pair in record C. The other important function associated with a record in BDFL is the record constructor function. Though the language permits certain syntactic variations the basic constructor function is written as

< a1:A1, a2:A2, . . ., an:An >,

where ai, i = 1 to n are the selectors and Ai, i = 1 to n are either the values or the names of data-lines on which the values arrive.

The dataset is an unordered collection of records. The concept of dataset closely follows the ledgers and tables that find extensive use in manual data processing. A new dataset may be created by one of the following ways.

a) Copying an existing dataset.

b) Collecting a sequence of records (of same type).

c) Taking a union of a sequence of datasets (of same type).

It may be emphasised that while (a) and (b) above correspond closely to the manner tables are filled, (c) mimics a collection of tables, namely, the ledger.

3.1 EXPRESSIONS IN BDFL

As the syntax and meaning of arithmetic expressions is universally accepted, we adopt it in BDFL as an atomic block. The syntax and semantics of Block expressions, Conditional expressions and loop expressions in BDFL follow, except for minor variations, those of Id [AGP80]. We give below one example of each of the above expressions to give a flavour of the syntax and semantics of BDFL expressions.

Example 1: It is desired to compute the net requirement for a part (QTY), by subtracting the quantity available in the stocks (FREE-QTY) from the quantity needed to start the production. The record carrying information about the quantity needed is available on data-line PART-PROD-ST-SCH.

Rule (3.1.1) shows a BDFL rule to obtain the QTY and to update the new FREE-QTY to zero. The values returned by the functions RETURN are transferred onto the data-lines named to the right of the symbol -->. The matching is based on the order of their occurrence.

```
[ ( 0 ) --> new FREE-QTY ;
  ( QTY of PART-PROD-ST-SCH  -  FREE-QTY )
                                --> QTY ;
    return QTY ; return new FREE-QTY ]
              --> QTY , new FREE-QTY
                                    (3.1.1)
```

NOTE : The use of lower case letters for key-words in the expressions is only for reading convinence.

Example 2: This example expands the specifications given for the previous example. Now, it is required to add a guard against the situation where FREE-QTY may exceed the need. The new version of the rule is shown below.

```
[ if ( FREE-QTY > QTY of PART-PROD-ST-SCH )
then
  ( FREE-QTY - QTY of PART-PROD-ST-SCH )
                        --> new FREE-QTY ;
  () --> QTY ;
  return QTY ; return new FREE-QTY
else
  (0) --> new FREE-QTY ;
  ( QTY of PART-PROD-ST-SCH - FREE-QTY )
                                --> QTY ;
  return QTY ; return new FREE-QTY
fi ] --> QTY , new FREE-QTY ;
                                        (3.1.2)
```

Example 3: It is desired to create a table giving the number of years (upto 9) and compound interest on an amount of Rs.1000 at the rate of 6.0% annually. The BDFL specification is given as expression (3.1.3).

```
[ initial
        ( 1000 ) --> AMOUNT ;
        ( 0 ) --> INTEREST ;
        ( 1 ) --> YEAR ;
  while ( YEAR < 10 ) do
        (( AMOUNT ) * 106 / 100 )
                --> new AMOUNT ;
        ( new AMOUNT - AMOUNT )
                --> new INTEREST ;
        ( YEAR + 1 ) --> new YEAR ;
  return set < YEAR : new YEAR,
               INTEREST : new INTEREST >
od ]
                                        (3.1.3)
```

Expression (3.1.3) also illustrates the use of the record constructor and the set constructor functions in BDFL. The function SET used in the RETURN clause returns a dataset created by collecting all its argument records.

3.2 DATASET MANIPULATION

A set of five functional forms have been defined in BDFL to permit manipulations involving datasets. These functional forms can be recursively nested to specify the operations involving more than one dataset.

The inclusion of these functional forms is inspired by the observation of the manual methods used in table and ledger handling.

Each of these functional forms define a block to which is made available a copy of the dataset. In the interior of these blocks either a single (subordinate) block or a sequence of them (depending on the functional form) exist. The functional makes available to these subordinate blocks, a record, a pair of records, or a dataset composed of the records from the incoming dataset, satisfying some (user defined) property.

For the purpose of their description we divide these functional forms into three groups. The following conventions will be used to describe the syntax of these functional forms.

- Keywords in functional forms are written in lower case.
- The alternative clauses are separated by ! symbol and enclosed in a pair of braces ({}).
- Zero or more occurrences are denoted by {}...; while, {}___ denotes 1 or more occurrences.

3.2.1 FOR EACH BLOCKS

This class of the functional forms were motivated by the need to apply a function to each entry in a table or to each sub-table in it.

```
[ {initial
  {<RULE>}___}
  for each set {<IDENTIFIER>! }
          in <DATASET-NAME>
          {with same {<SELECTOR>
          {ASCENDING!DESCENDING! }}___}
do
  {<RULE>}___
  <RETURN CLAUSES>
od ]
```

(a)

```
[ {initial
  {<RULE>}___}
  for each record {<IDENTIFIER>! }
          in <DATASET-NAME>
          {{ascendins!descendins}
          {<SELECTOR>}___}...
          {while <BOOLEAN EXPRESSION>}
  do
    {<RULE>}___
    {<RETURN CLAUSE>!remainder}...
od ]
```

(b)

Fig. 1: The syntax for FOR EACH BLOCKS
(a) FOR EACH SET (b) FOR EACH RECORD

The syntax for the functional forms in this class is shown in Fig. 1. The functional FOR EACH SET partitions the input dataset based on the values associated with the specified selectors. These partitioned datasets are then distributed among the subordinate blocks. The ASCENDING/DESCENDING clause defines the interconnections between these subordinate blocks. When WITH SAME clause is omitted the partition of the dataset into internal datasets is based on the creation history of the dataset.

Example 4: Let f be a function that applies to a set of 'production start schedules' for a part, and to the set FREE-STOCKS. The function returns a dataset containing the requirements for the part.

Given an input dataset carrying 'part production start schedules' for various parts, the expression (3.2.1.1) produces a dataset containing part requirements for all parts in the input dataset.

```
[ for each set in PART-PROD-ST-SCH
            with same PART-NO
do
     f ( PART-PROD-ST-SCH, FREE-STOCKS)
                    --> PART-REQ-SUBSET ;
     return set PART-REQ-SUBSET
od ]
```

(3.2.1.1)

The functional FOR EACH RECORD also follows similar set of rules, with the major exception being that it makes available a record to the block that is subordinate to it. The key-word remainder in Fig. 1 (b) represent the function that in conjunction with the while clause returns a dataset comprising of unscanned records of input dataset.

Example 5: Let the data-line FREE-STOCKS provides a record containing information about the available stock of a part. Let s be the function as defined by Example 2.

Expression (3.2.1.2) uses the functional FOR EACH RECORD to extend the example to datasets. Thus, the expression (3.2.1.2) defines a function that returns a set of records giving net requirements (by date) for a part. The input to the function is set of projected needs (by date) for that part.

```
[ initial
  ( QTY of FREE-STOCKS ) --> FREE-QTY ;
for each record in PART-PROD-ST-SCH
        in ascending START-DATE
do
    s --> QTY, new FREE-QTY ;
    return set < PART-NO : PART-NO of
                          PART-PROD-ST-SCH ,
                 QTY : QTY ,
                 START-DATE : START-DATE of
                          PART-PROD-ST-SCH >
od ]
```

(3.2.1.2)

Fig. 2 shows an equivalent block diagram for the expression (3.2.1.2).

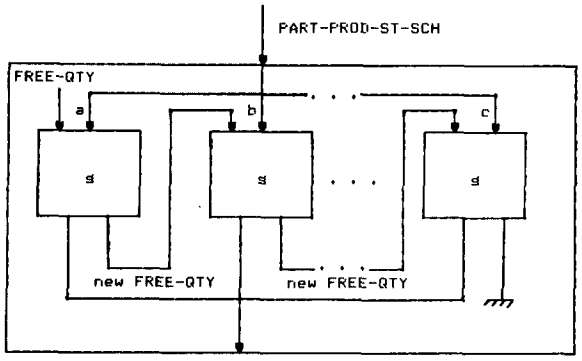

Fig.2 : The equivalent block diagram for expression (3.2.1.2). The lines labelled a,b,c will have the label PART-PROD-ST-SCH and satisfy START-DATE of a < START-DATE of b < START-DATE of c.

3.2.2 USING BLOCKS

The inclusion of these functional forms in BDFL is aimed at facilitating the specification of data processing activities that require the application of a function only to a class of records (or to a single record) in a dataset.

The syntax for the two functional forms in this class is given in Fig. 3. USING SET (RECORD) functional constructs for its subordinate blocks a dataset (record) satisfying the boolean expression indicated by the user. These functional forms resemble the conditional expressions in certain respects. The data is passed on to one of the two alternative functions (blocks) based on, whether the data bag is empty (i.e. no record satisfying the boolean expression existed) or not.

The dataset generator functions UPDATE and

APPEND have been introduced to be used in conjunction with these functional forms. The UPDATE returns a dataset that matches the incoming dataset in all entries except those that stisfy the boolean expression. The entries (entry) that satisfy the boolean expression are replaced by the argument of the function UPDATE. The function APPEND returns the dataset obtained by adding its argument to the incoming dataset.

```
[ {initial
          {<RULE>}____>
  using {set|record} {<IDENTIFIER>| }
        in <DATASET-NAME>
        with <BOOLEAN EXPRESSION>
  do
     {<RULE>}____
     {<RETURN CLAUSE>|<UPDATE CLAUSE>}____
  { |not found :
     {<RULE>}____
     {<RETURN CLAUSE>|<APPEND CLAUSE>}____>
od ]
```

Fig. 3: The syntax for USING BLOCKS

Example 6: Given a set of the production start schedules for a part, it is desired to obtain the stock position for the part. The stock position is then used to compute the net purchase requirements for the part. Let, h be the function that returns the purchase requirements given the necessary data.

Expression (3.2.2.1) uses the function h to complete the BDFL specification for the desired function.

```
[ using record in FREE-STOCKS
     with PART-NO = PART-NO of
                          PART-PROD-ST-SCH
do
   h --> PART-REQ-SUBSET ;
   return PART-REQ-SUBSET
od ]
                                   (3.2.2.1)
```

3.2.3 FOR EACH TUPLE BLOCK

This functional has been introduced in the language to permit processing of pairs of records drawn from two datasets. The criterion for the pairing of the records is indicated by the user as he specifies the selectors that should have matching values associated with them. Any record (irrespective of dataset it belongs to), that does not have a matching record in other dataset is paired with a null record.

A special case of this functional form that find extensive use in business data processing applications is updating of master file by a transaction file. The complete syntax for the functional form is given in Fig. 4. Other considerations are similar to those indicated for FOR EACH RECORD block.

```
[ {initial
       {<RULE>}____>
for each tuple {<IDENTIFIER>,<IDENTIFIER>|}
          in <DATASET-NAME>,<DATASET-NAME>
          in {{ascending|descending| }
            <LEFT ANGL. BRACKET>
               <SELECTOR>,<SELECTOR>
                  <RIGHT ANGL. BRACKET>}____
do
       {<RULE>}____
       <RETURN CLAUSES>
od ]
```

Fig. 4: The syntax for FOR EACH TUPLE BLOCK

Example 7: Expression (3.2.3.1) updates the master file MAST by the transaction file TRANS.

```
[ for each tuple TREC,MREC
    in TRANS,MAST
    in ascending < PART-NO, ITEM-NO >
do
  [ if ( null ( TREC ))
  then
       return MREC
  else
       ( QTY of MREC - QTY of TREC )
                              --> QTY ;
       return < ITEM-NO : ITEM-NO of MREC ,
                QTY : QTY >
  fi ] --> REC ;
  return set REC
od ]
                                   (3.2.3.1)
```

4. A TRANSLATOR FOR BDFL

A multiphase translator has been implemented to permit automatic generation of COBOL programs corresponding to BDFL specifications. The translator has been written in PASCAL and is portable.

We present here only a sketchy description of the translation algorithm. The details can be found in [MAL81]. A description of

the major steps in the translation process follows:

i. Each block in the specification is assigned a unique name. The unique name chosen permits a simple method for deducing the name of the block that contains it. Hence, it is possible to reconstruct the nested structure of the BDFL blocks any time during the translation process.

ii. The data-lines implied by the specification are identified. To avoid possible conflicts the data-line names are made unique by adding necessary prefixes where needed.

The data lines that are of interest for the purpose of translation can be grouped into two classes.

- The data-lines that represent the flow of information at a given level of nesting. These data-lines are identified by providing a data-line from the source of the data to the block that needs it. The data needs that do not have the corresponding source at that level of the nesting becomes the data need at the next higher level.

- The other class of data-lines deals with the data-lines that carry information from an outer block to inner blocks. The Loop expression, Conditional expression and dataset manipulation functional forms that bring the need for considering these data-lines are assumed for the purpose of this step to be responsible for making necessary provisions to create these data-lines.

The data-lines that do not have any source even at the highest level represent the inputs to the specification. The outputs, usually get defined by the return clauses in the outermost block.

iii. Define a directed graph with the expression names as nodes and the data-lines as (directed) arcs. Topologically sort the blocks at each level (i.e. the blocks that are contained in a single block). Rearrange, at all levels, the blocks into their topological order.

The purpose of this re-arrangement of the blocks is to convert the BDFL specification describing the processing activities being carried out in parallel and distributed in space, into a computation that is to be carried by a single processor and extends in time instead of space. The topological ordering of the expressions permit sequential execution without ever having to starve for data.

iv. Taking advantage of the arrangement obtained in (iii) above, deduce the type for each data-line. Some of the rules used for deducing types of the data-line are :

- Data-line receiving the result of an arithmetic computation is assigned the type integer.

- Data-line that receives a value from another data-line is grouped with the donor data-line for the purpose of their type deduction.

- An application of the projection function to the value on a data-line implies its type to be other than scalar.

- A use of record constructor and set constructor reveals the type of data-line receiving the value.

- A use of dataset manipulation functional form also reveals the type for the data-lines incident to it and the data-lines inside the block.

For the data-lines of type record and data-set lists of the 'selectors' associated with the records are also prepared.

v. Using information collected in (iv), generate the SCHEMA for the DBMS. For this purpose, record-types are defined for inputs, outputs and other records and datasets. Beside these a few more record-types are defined to perform housekeeping chores.

Based on the usage patterns (in the BDFL specification) for the data-lines of the type dataset the (CODASYL) sets are defined in the SCHEMA. The definition of these sets is guided by the desire to be able to format the dataset as soon as it is created, in a form that is conducive to the exploitation by the functional forms for dataset manipulation, to which the dataset is destined to be put.

vi. The COBOL code is generated in the following fashion.

Each block results in a sequence of consecutive paragraphs in the COBOL code. The necessary code to 'run' the block is produced; while PERFORM statements are introduced to run the subordinate blocks.

5. AN EXAMPLE

Fig. 5 presents the overview of an example. The example owes its origin to Clifton [CLIF71].

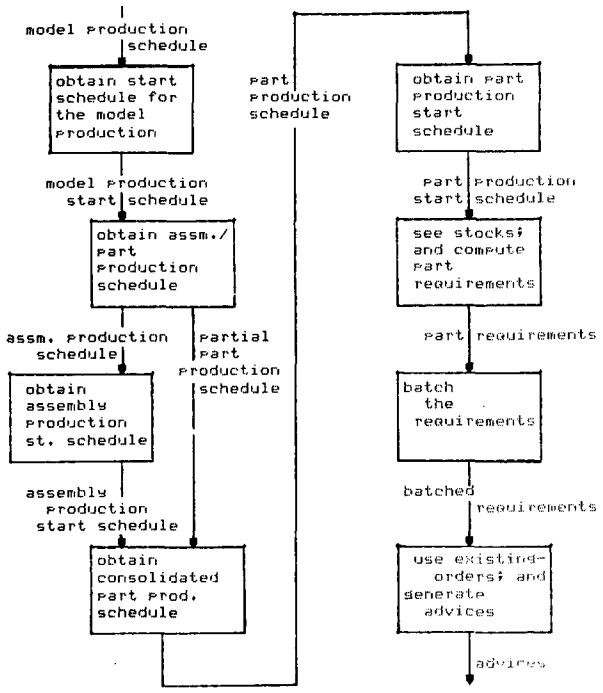

Fig. 5 : Overview of a business data processing application

This example concerns a manufacturing company engaged in manufacturing a range of products. A marketable product, called model, is built of a number of parts that the company, either procures in the market or produces locally. To facilitate the production, a range of assemblies are also produced. These assemblies go into the production of the models and are produced only when demand arises.

It is desired to automate the process of order placement and cancellation for the part based on the projected demand for the models.

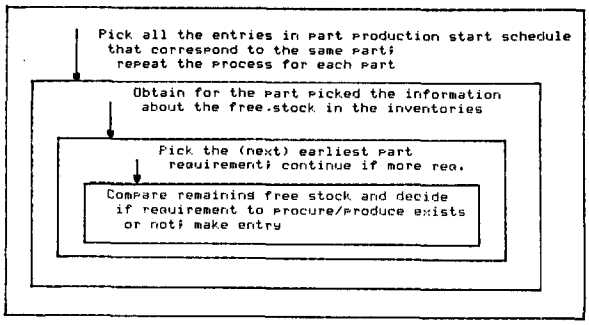

Fig. 6 : The details for net part requirement computation

Here we only take the details of a function that enables the calculation of the requirements for the various parts. Fig. 6 describes, in detail, the flow of information from the function to its subfunctions and between them. Bits of this specification have already been converted to BDFL specifications in Section 3. The consolidated BDFL specification for the function appears in Fig. 7.

The COBOL program for the example was generated. The program was about 2900 lines long. The program executed correctly for all the data it was tried on.

The generated COBOL programs show some undesireable traits too. Firstly, the object program size tends to be large even for small specifications. The ratio of the object program size to the source BDFL specification size improves as the applications become more involved and complex. The reason for the unduly large COBOL code, at the lower end of the applications, lies in the fact that the generated program tends to be dominated by the code responsible for setting-up the DBMS for input and output data. The housekeeping chores that find their applications in more involved programs, are also performed in these simple cases even though they do not find any use in these applications.

A number of modifications can be made to compact the generated code. The compaction, besides cutting the run-time makes the program readable.

One of the compaction strategies involves deleting the 'MOVE CORR' statements that deal in records with no common field. Further, the values of some of the identifiers can be computed statically (at

compile time). Thus the unwarranted computations and, if desired, these identifiers can be completely removed from the programs. Elimination of dead assignments also is expected to provide much needed compaction.

By assigning user specified names to BDFL blocks, and hence to paragraphs in the generated programs (at present these names are generated automatically), the readability can be improved further.

The use of data base currency-flow analysis on the lines of data-flow analysis [see e.g. AHU77] is expected to significantly improve the program efficiency.

The transformations [IPTC76] at the BDFL level can be made to convert the specifications into a form that leads to efficient object code without affecting the meaning of the specification.

```
[ for each set in PART-PROD-ST-SCH
    with same PART-NO
do
   [ using record in FREE-STOCKS
      with PART-NO = PART-NO of
                          PART-PROD-ST-SCH
   do
      [ initial
          ( QTY of FREE-STOCKS )
                  --> FREE-QTY ;
      for each record in PART-PROD-ST-SCH
                in ascending START-DATE
      do
         [ if ( FREE-QTY > QTY of
                        PART-PROD-ST-SCH )
         then
           ( FREE-QTY - QTY of
                        PART-PROD-ST-SCH )
                        --> new FREE-QTY ;
           ()  --> QTY ;
           return QTY ;
           return new FREE-QTY
         else
           (0)  --> new FREE-QTY ;
           ( QTY of PART-PROD-ST-SCH -
                    FREE-QTY )  --> QTY ;
           return QTY ;
           return new FREE-QTY
        fi ]  --> QTY , new FREE-QTY ;
        return set < PART-NO : PART-NO of
                          PART-PROD-ST-SCH ,
                     QTY : QTY ,
                     START-DATE :
                         START-DATE of
                         PART-PROD-ST-SCH >
     od ]  --> PART-REQ-SUBSET ;
     return PART-REQ-SUBSET
   od ]  --> PART-REQ-SUBSET ;
   return set PART-REQ-SUBSET
od ]
```

Fig.7: An example of BDFL specification.

6. CONCLUSIONS

In this paper a powerful language to write the specifications of business data processing activities at a very high level is defined. The language is versatile enough to be of use in specifying most of the data processing applications that are of interest to business organisations. Since we aimed at providing a language that even a non-programming user can easily learn, many advanced issues in system design, e.g. exception handling, integrity checking, recovery etc. have not been explored yet.

The use of DBMS has eliminated the need for an explicit design of the data base for the application. The feasibility of the language and its conversion to a procedural description has been shown by actually designing and implementing the translator.

The procedural representation obtained through automatic translation is at present not very efficient and will benefit by human interventions (say, to tune the DBMS). We are, however, confident that the efficiency of the system will improve as our understandings of the system matures and most certainly with the availability of data-flow machines.

ACKNOWLEDGEMENTS: The authors gratefully acknowledge the help given by Prof. K.V.Nori in implementing the program generator. Prabhakar and Harish Karnick are responsible for many suggestions that made this report presentable. Even though we have not been able to accomodate all suggestions and advices given by the referees, their help is gratefully acknowledged.

REFERENCES

[AGP78] Arvind, K.P.Gostelow and W.Plouffe, An asynchronous Programming Language and Computing Machine* , Tech. Rep. 114A, Dept. of Info. and Comp. Sc., UCI, Irvine (Dec. 78).

[AGP80] Same as [AGP78], revised in (June 80).

[AHU77] Aho,A.V. and J.D.Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Ma.,1977.

[BAC78] Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Comm. ACM 21,8 (Aug. 78).

[CLIF71] Clifton, H.D., Data Processing System Design, Business Book Limited,1971.

[DEN73] Dennis, J.B., First Version of a Data Flow Procedure Language. Comp. Structure group memo 93, Lab. for Comp. sc., Cambridge, Massachusetts (Nov. 73).

[HHKW77] Hammer, M., W.G.Howe, V.J.Kruskal and I.Wladawsky, A Very High Level Programming Language for Data Processing Applications, Comm. ACM 20,11 (Nov. 77).

[IFTC76] Standish, T.A., D.C.Harriman, D.F.Kibler and J.M.Neighbors, The Irvine Program Transformation Catalogue, Dept. of Info. and Comp. Sc., UCI, Irvine (Jan.1976).

[MAL81] Malhotra, V.M., A Data-Flow Based Specification Language and its Translator for Business Applications, Ph. D. Thesis, Comp. Sc. Prog., Indian Inst. of Tech., Kanpur, Aug. 81.

[PPS79] Prywes, N.S., A.Pnueli and S.Shastry, Use of a Nonprocedural Specification Language and Associated Program Generator in Software development, TOPLAS 1,2 (Oct.79).